Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

імені Ігоря Сікорського»

Факультет прикладної математики

Кафедра програмного забезпечення комп’ютерних систем

КУРСОВА РОБОТА

з дисципліни «Об’єктно-орієнтоване програмування»

на тему

Шаблони проектування в ООП. Графічний редкатор

Виконав студент

ІІ курсу групи КП-73

Булаєвський Ігор Олегович

залікова книжка КП-7303

Керівник роботи

доц., к.т.н. Заболотня Т.М.

Оцінка

_____

(дата, підпис)

Київ-2019

ЗМІСТ

ВСТУП

Дана курсова робота присвячена розробці програмного забезпечення графічного редактора за допомогою використання шаблонів проектування. Задача моделювання програмного забезпечення графічного редактора є досить актуальною, адже графічний редактор – звична річ, яка зазвичай інстальована на всіх комп’ютерах, і для комфортної роботи користувача зі звичним інтерфейсом графічного редактора на різних платформах, необхідно розробити певні рішення, які представлять основні потоки даних зображень, з якими взаємодіють користувачі під час під час редагування улюблених фото. Дана тематика вибрана для курсової роботи тому, що результати абстрагування об’єктів у даній спроектованій системі дозволяють застосувати всі вивчені методи та принципи об’єктно-орієнтованого програмування для створення програмного забезпечення, зокрема мати змогу правильно організувати код за допомогою шаблонів проектування.

Об’єктом курсової роботи є *графічний редактор*.

Метою роботи є *розроблення програмного забезпечення для графічного редактора* з використанням шаблонів проектування.

Для досягнення описаної мети необхідно виконати такі **пункти завдань**:

- Абстрагувати об’єкти предметної галузі;
- Розробити структурну організацію програмного забезпечення за допомогою використання основних принципів об’єктно-орієнтованого програмування та шаблонів проектування;
- Визначити та описати функціональні характеристики програми;
- Обґрунтувати вибір шаблонів проектування;
- Розробити графічний інтерфейс користувача;
- Виконати реалізацію програмного забезпечення відповідно до технічного завдання;

- Виконати тестування розробленої програми;
- Оформити документацію з курсової роботи.

Розроблене програмне забезпечення складається з таких логічних частин: інтерфейсу користувача, накладання на зображення різних фільтрів, модуля, що відповідає за виконання над зображенням маніпуляцій зміни розміру/повороту/ відображення.

Використані шаблони проектування: **Adapter, Strategy, Prototype, Proxy (**у вигляді Protection Proxy**), Flyweight, Command, State.**

Розроблене програмне забезпечення може бути використане як частина пре-інстальованих на ПК програм, які постачаються сумісно при покупці нового гаджета.

Пояснювальна записка складається зі вступу, двох розділів, загальних висновків та списку використаних джерел (*три* найменування). Робота містить __ рисунків. Загальний обсяг роботи – __ друкованих сторінок.

1. ОПИС СТРУКТУРНО-ЛОГІЧНОЇ СХЕМИ ПРОГРАМИ

1.1. Модульна організація програми

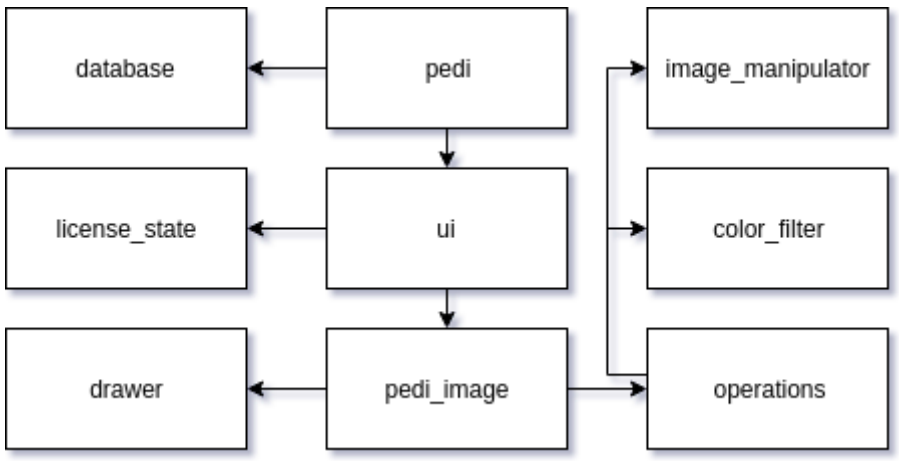

Рис. 1.1.1. Модульна організація програми

pedi – точка входу в виконання програми. Забезпечує запуск графічного інтерфесу. Зв’язується з базою даних.

ui – модуль графічного інтерфейсу. Містить в собі класи, які є сотностями вкладок редагування зображень (застосування фільтрів, зміна параметрів зображення, зміна розміру зображення, поворот/відображення зображення).

pedi_image – модуль, який працює з основною сутністю зображення, відповідає за інтерфейс для створення, копіювання, редувагування та збереженя зображення.

drawer – модуль, який відповідає за відображення на екрані зображення.

operations – модуль, який укамулює і собі усі операції з інших модулів, які дозволяють будь-яким чином редагувати зображення.

color_filter – модуль, який містить у собі логіку накладання на зображення фільтрів, самостійно редагуючи пікселі зображення.

image_manipulator — модуль, який забезпечеє операції над зображення, для яких використовуються посторонні бібліотеки.

license_state — модуль, який описує логіку поведінки програми в залежності від стану її ліцензійної активації.

Database — модуль, який явлає собою сховище існуючих користувачів.

1.2. **Функціональні характеристики**

Робота з програмою розпочинається з вікна авторизації користувача. Користувач має змогу авторизуватися або пропустити цей крок.

З вікна авторизації користувач потрапляє у вікно головного інтерфейсу користувача, де зразу він може за допомогою кнопки Upload обрати зображення з файлової системи для подальшого його редагування.

Як тільки було обране перше зображення на вікні головного інтерфейсу з’являються чотири вкладки, на яких операції редагування посортовані за логічною ідентичністю:

- *FiltersTab* — на цій вкладці можна застосувати до зображення один з трьох фільтрів. Кнопки кожного фільтру містять у собі прев’ю того, як зображення виглядатиме у разу застосування фільтру до нього.

- AdjustingTab — ця вкладка містить повзунки для редагування трьох характеристик зображення: яскравість, контраст так чіткість.

- *ModificationTab* — вкладка, на якій можна змінити розмір зображення.

- *RotationTab* — вкладка, на якій користувач може повернути зображення за/против годиниковою стрілкою, відобразити зображення зліва направо/зверху вниз.

Відредагувавши сображення користувач може відмінити усі зміни натиснувши кнопку *Reset.* При натисканні кнопки *Save* існує два сценарії:

- якщо програма активована, то пропонується вибрати місце, куди зберегти зображення;

- якщо неативована, то відображається повідомлення про неможливість виконати дану операцію.

1.3. Опис реалізованих класів

1. **ColorFilter –** клас, який реалізує обробку зображення при накладанні на нього того чи іншого фільтру. Містить класи **SepiaFilter, BlackWhiteFilter, NegativeFilter.**

color_filter.py

```
class ColorFilter(ABC):
    @abstractmethod
    def apply(self, pedi_image):
        pass


class SepiaFilter(ColorFilter):
    def apply(self, pedi_image):
        img = pedi_image.qimage
        pix = img.load()
        for i in range(img.width):
            for j in range(img.height):
            s = sum(pix[i, j]) // 3
            k = 30
            pix[i, j] = (s+k*2, s+k, s)


class BlackWhiteFilter(ColorFilter):
    def apply(self, pedi_image):
        img = pedi_image.qimage
        pix = img.load()
        for i in range(img.width):
            for j in range(img.height):
                s = sum(pix[i, j]) // 3
                pix[i, j] = (s, s, s)


class NegativeFilter(ColorFilter):
    def apply(self, pedi_image):
        img = pedi_image.qimage
        pix = img.load()
            for i in range(img.width):
                for j in range(img.height):
                    pix[i, j] = (255 - pix[i, j][0], 255 -
                    pix[i, j][1], 255 - pix[i, j][2])
```

2. Database – клас для імітації роботи бази даних. Реальна БД була замінена її імітацією тому що головною задачею графічного редактора є саме редагування зображень, а не робота з користувачами.

Містить власне клас **User.**

database.py

```
class User:
    def __init__(self, login):
        self.login = login

    def __eq__(self, other):
        return self.login == other

class DataBase:
    __users = [
        User('admin'),
        User('tmp_user'),
        User('me')
    ]

    @staticmethod
    def get_user(login):
        try:
            index = DataBase.__users.index(login)
            return DataBase.__users[index]
        except:
            return None
```

3. Drawer – клас, який відповідає за відображення на екрані зображення різними способами. Є членом паттерну “Команда”.

drawer.py

```
class Drawer:
    def set_operation(self, pedi_image, operation):
        pedi_image.operation = operation

    def process(self, pedi_image, option):
        pedi_image.redraw(option)
```

4. ImageManipulator — клас, похідні якого утворюють повну множину усіх операцій, які можна виконати над зображеннями. Містить класи **ImageResizer , ImageResizer, ImageRotator, ImageFliper, ImageFilterer, ImageBrightener, ImageContraster, ImageSharpner.**

Є членом поттерну “Легковаговик” та “Адаптер”.

image_manipulator.py

```
class ImageManipulator(ABC):
    @abstractmethod
    def execute(self, pedi_image, option):
        pass

class ImageResizer(ImageManipulator):
    def execute(self, pedi_image, option):
        pedi_image.qimage = pedi_image.qimage.resize(option)

class ImageRotator(ImageManipulator):
    def execute(self, pedi_image, option):
        angle = option

        pedi_image.qimage = pedi_image.qimage.rotate(angle, expand=True)

class ImageFliper(ImageManipulator):
    def execute(self, pedi_image, option):
        direction = option

        pedi_image.qimage = pedi_image.qimage.transpose(direction)

class ImageFilterer(ImageManipulator):
    __filters = {
        'sepia': SepiaFilter(),
        'black_white': BlackWhiteFilter(),
        'negative': NegativeFilter()
    }

    def execute(self, pedi_image, option):
        filter_name = option
        concrete_filter = ImageFilterer.__filters[filter_name]

        concrete_filter.apply(pedi_image)

class ImageBrightener(ImageManipulator):
    def execute(self, pedi_image, option):
        factor = option

        if factor > BRIGHTNESS_FACTOR_MAX or factor < BRIGHTNESS_FACTOR_MIN:
            raise ValueError("factor should be [0-2]")

        enhancer = ImageEnhance.Brightness(pedi_image.qimage)
        pedi_image.qimage = enhancer.enhance(factor)

class ImageContraster(ImageManipulator):
    def execute(self, pedi_image, option):
        factor = option

        if factor > CONTRAST_FACTOR_MAX or factor < CONTRAST_FACTOR_MIN:
            raise ValueError("factor should be [0.5-1.5]")
```

```
        enhancer = ImageEnhance.Contrast(pedi_image.qimage)
        pedi_image.qimage = enhancer.enhance(factor)

class ImageSharpner(ImageManipulator):
    def execute(self, pedi_image, option):
        factor = option

        if factor > SHARPNESS_FACTOR_MAX or factor < SHARPNESS_FACTOR_MIN:
            raise ValueError("factor should be [0.5-1.5]")

        enhancer = ImageEnhance.Sharpness(pedi_image.qimage)
        pedi_image.qimage = enhancer.enhance(factor)
```

5. State - клас, який описує поведінку програми на натисненні кнопки *Save*. В залежності від статусу активації використовується один з двох класів **LicensedState/NotLicensedState**. Є членом паттерну “Стан”.

license_state.py

```
class State(ABC):
    @staticmethod
    def run(parent):
        pass

class LicensedState(State):
    @staticmethod
    def run(parent):
        logging.debug('on licensed save state')

        path, _ = QFileDialog.getSaveFileName(parent.parent, "QFileDialog.getSaveFileName()",
f"pedi_{parent.file_name}",
"Images (*.png *.jpg)")
        return path

class NotLicensedState(State):
    def run(parent):
        logging.debug('on not licensed save state')

        QMessageBox.warning(parent.parent, 'Error',
            "Login to unlock this feature")
```

6. Operations — клас, який є фабрикою для паттерну “Легковаговик”. Забезпечує доступ до об’єктів конкретних операцій.

operations.py

```
from image_manipulator import *

class Operations:
    __all_operations = {
        'resize': [ImageResizer, None],
        'rotate': [ImageRotator, None],
        'flip': [ImageFliper, None],
        'filter': [ImageFilterer, None],
        'brightness': [ImageBrightener, None],
        'contrast': [ImageContraster, None],
        'sharpness': [ImageSharpner, None]
    }

    def __has_operation(self, name):
        return bool(Operations.__all_operations[name][1])

    def __create_operation(self, name):
        instance = Operations.__all_operations[name][0]()
        Operations.__all_operations[name][1] = instance

    def __get_instance(self, name):
        return Operations.__all_operations[name][1]

    __filters_name = ['sepia', 'black_white', 'negative']

    def get_operation(self, name):
        if (not self.__has_operation(name)):
            self.__create_operation(name)
        return self.__get_instance(name)
```

7. PediImage — клас, який відповідає за основу сутність програми. Описує зображення та надає інтерфейс для роботи з ним. Є учасником паттернів “Сратегія”, “Прототип” та “Команда”. У файлі також описані клас **DisplayStrategy** та похідні від нього **LabelStrategy** та **ThumbStrategy**, які є учансниками паттерну “Стратегія” та відповідаються за способи відображення зображення на екрані.

pedi_image.py

```
class DisplayStrategy(ABC):
    def __init__(self, parent):
        self.parent = parent

    @abstractmethod
    def display(self, pedi_image):
        pass

class LabelStrategy(DisplayStrategy):
    def display(self, pedi_image):
        pixmap = ImageQt.toqpixmap(pedi_image.qimage)

        self.parent.image_label.setPixmap(pixmap)

        logging.debug('image label updated')

class ThumbStrategy(DisplayStrategy):
    def __init__(self, parent):
        super().__init__(parent)
        self.__filterer = ImageFilterer()

    def __display_one(self, thumb, pedi_image):
        image_filtered = pedi_image.clone()
        self.__filterer.execute(image_filtered, thumb.name)

        preview_pixmap = ImageQt.toqpixmap(image_filtered.qimage)
        thumb.setPixmap(preview_pixmap)

        logging.debug('%s thumb updated' % thumb.name)

    def display(self, pedi_image):
        for thumb in self.parent:
            thread = Thread(target=self.__display_one,
args=(thumb, pedi_image))
            thread.start()

class ICloneable():
    @abstractmethod
    def clone(self):
        pass

class PediImage(ICloneable):
    def __init__(self, img_path):
```

```
        pixmap = QPixmap(img_path)
        self.qimage = ImageQt.fromqpixmap(pixmap)

        self.strategy = None

    @property
    def width(self):
        return self.qimage.width

    @property
    def height(self):
        return self.qimage.height

    def set_display_strategy(self, strategy):
        self.strategy = strategy

    def redraw(self, option):
        if (not self.operation):
            return
        self.operation.execute(self, option)

    def show(self):
        if (self.strategy):
            self.strategy.display(self)

    def clone(self):
        old_qimage = self.qimage
        qimage_copy = self.qimage.copy()
        self.qimage = None

        self_copy = copy(self)
        self.qimage = old_qimage

        self_copy.qimage = qimage_copy
        return self_copy

    def save(self, new_img_path):
        self.qimage.save(new_img_path)
```

8. ActionTabs – клас, який відповідає за частину графічного інтерфуйсу, яка містить у собі вкладки з категоріями операцій над зображеннями.

ui.py

```
class ActionTabs(QTabWidget):
    def __init__(self, parent):
        super().__init__()
        self.parent = parent

        self.filters_tab = FiltersTab(self)
        self.adjustment_tab = AdjustingTab(self)
```

```
        self.modification_tab = ModificationTab(self)
        self.rotation_tab = RotationTab(self)

        self.addTab(self.filters_tab, "Filters")
        self.addTab(self.adjustment_tab, "Adjusting")
        self.addTab(self.modification_tab, "Modification")
        self.addTab(self.rotation_tab, "Rotation")

        self.setMaximumHeight(190)
```

9. FiltersTab – вкладка, яка відповідає за частину графічного інтерфейсу, де відображається прев'ю зображень у разу застосування до них кольорного фільтру.

ui.py

```
class FiltersTab(QWidget):
    def __init__(self, parent):
        super().__init__()
        self.parent = parent

        self.main_layout = QHBoxLayout()
        self.main_layout.setAlignment(Qt.AlignCenter)

        self.thumbs = [
            self.create_filter_thumb('sepia'),
            self.create_filter_thumb('black_white'),
            self.create_filter_thumb('negative')
        ]
        global strategies
        strategies['thumb'] = ThumbStrategy(self.thumbs)

        self.setLayout(self.main_layout)

    def create_filter_thumb(self, name):
        thumb_label = QLabel()
        thumb_label.name = name
        thumb_label.setStyleSheet("border:2px solid #ccc;")

        thumb_label.setToolTip(f"Apply <b>{name}</b> filter")

        thumb_label.setCursor(Qt.PointingHandCursor)
        thumb_label.setFixedSize(THUMB_SIZE, THUMB_SIZE)
        thumb_label.mousePressEvent = partial(self.on_filter_select, name)

    self.main_layout.addWidget(thumb_label)
        return thumb_label

    def on_filter_select(self, name, e):
        logging.debug('%s filter selected' % name)
```

```
        global image_preview, drawer, operations
        operation = operations.get_operation('filter')
        drawer.set_operation(image_preview, operation)
        drawer.process(image_preview, name)
        self.parent.refresh_image()
```

10. AdjsutingTab – вкладка, яка відповідає за частину графічного інтерфейсу, де можна змінювати яскравість, контраст та чіткість зображення.

ui.py

```
class AdjustingTab(QWidget):
    def __init__(self, parent):
        super().__init__()
        self.parent = parent

        contrast_label = QLabel("Contrast")
        contrast_label.setAlignment(Qt.AlignCenter)

        brightness_label = QLabel("Brightness")
        brightness_label.setAlignment(Qt.AlignCenter)

        sharpness_label = QLabel("Sharpness")
        sharpness_label.setAlignment(Qt.AlignCenter)

        contrast_slider = QSlider(Qt.Horizontal, self)
        contrast_slider.setMinimum(SLIDER_MIN_VAL)
        contrast_slider.setMaximum(SLIDER_MAX_VAL)
        contrast_slider.sliderReleased.connect(
            self.on_contrast_slider_released)
        contrast_slider.setToolTip(str(SLIDER_MAX_VAL))

        brightness_slider = QSlider(Qt.Horizontal, self)
        brightness_slider.setMinimum(SLIDER_MIN_VAL)
        brightness_slider.setMaximum(SLIDER_MAX_VAL)
        brightness_slider.sliderReleased.connect(
            self.on_brightness_slider_released)
        brightness_slider.setToolTip(str(SLIDER_MAX_VAL))

        sharpness_slider = QSlider(Qt.Horizontal, self)
        sharpness_slider.setMinimum(SLIDER_MIN_VAL)
        sharpness_slider.setMaximum(SLIDER_MAX_VAL)
        sharpness_slider.sliderReleased.connect(
            self.on_sharpness_slider_released)
        sharpness_slider.setToolTip(str(SLIDER_MAX_VAL))

        self.sliders = {
            'brightness': brightness_slider,
            'contrast': contrast_slider,
            'sharpness': sharpness_slider
        }
```

```
        main_layout = QVBoxLayout()
        main_layout.setAlignment(Qt.AlignCenter)

        main_layout.addWidget(contrast_label)
        main_layout.addWidget(contrast_slider)

        main_layout.addWidget(brightness_label)
        main_layout.addWidget(brightness_slider)

        main_layout.addWidget(sharpness_label)
        main_layout.addWidget(sharpness_slider)

        self.reset_sliders()
        self.setLayout(main_layout)

    def reset_sliders(self):
        self.sliders['brightness'].setValue(SLIDER_DEF_VAL)
        self.sliders['sharpness'].setValue(SLIDER_DEF_VAL)
        self.sliders['contrast'].setValue(SLIDER_DEF_VAL)

    @staticmethod
        def _get_converted_point(user_p1, user_p2, p1, p2, x):
        r = (x - user_p1) / (user_p2 - user_p1)
        return p1 + r * (p2 - p1)

    def __adjust(self, prop):
        logging.debug('on %s slider released' % prop)
        slider = self.sliders[prop]

        slider.setToolTip(str(slider.value()))

        factor = AdjustingTab._get_converted_point(SLIDER_MIN_VAL, SLIDER_MAX_VAL, BRIGHTNESS_FACTOR_MIN,
BRIGHTNESS_FACTOR_MAX, slider.value())
        logging.debug("%s factor: %f" % (prop, factor))

        global image_preview, drawer, operations
        operation = operations.get_operation(prop)
        drawer.set_operation(image_preview, operation)
        drawer.process(image_preview, factor)
        self.parent.parent.refresh_image()

    def on_brightness_slider_released(self):
        self.__adjust('brightness')

    def on_contrast_slider_released(self):
        self.__adjust('contrast')

    def on_sharpness_slider_released(self):
        self.__adjust('sharpness')
```

11. ModificationTab – вкладка, яка відповідає за частину графічного інтерфейсу, де можна змінювати розмір зображення у пікселях.

ui.py

```
class ModificationTab(QWidget):
    """Modification tab widget"""

    def __init__(self, parent):
        super().__init__()
        self.parent = parent

        self.width_label = QLabel('width:', self)
        self.width_label.setFixedWidth(100)

        self.height_label = QLabel('height:', self)
        self.height_label.setFixedWidth(100)

        self.width_box = QLineEdit(self)
        self.width_box.textEdited.connect(self.on_width_change)
        self.width_box.setMaximumWidth(100)

        self.height_box = QLineEdit(self)
        self.height_box.textEdited.connect(self.on_height_change)
        self.height_box.setMaximumWidth(100)

        self.unit_label = QLabel("px")
        self.unit_label.setMaximumWidth(50)

        self.apply_buttton = QPushButton("Apply")
        self.apply_buttton.setFixedWidth(90)
        self.apply_buttton.clicked.connect(self.on_apply)

        width_layout = QHBoxLayout()
        width_layout.addWidget(self.width_box)
        width_layout.addWidget(self.height_box)
        width_layout.addWidget(self.unit_label)

        apply_layout = QHBoxLayout()
        apply_layout.addWidget(self.apply_buttton)
        apply_layout.setAlignment(Qt.AlignRight)

        label_layout = QHBoxLayout()
        label_layout.setAlignment(Qt.AlignLeft)
        label_layout.addWidget(self.width_label)
        label_layout.addWidget(self.height_label)

        main_layout = QVBoxLayout()
        main_layout.setAlignment(Qt.AlignCenter)

        main_layout.addLayout(label_layout)
        main_layout.addLayout(width_layout)
        main_layout.addLayout(apply_layout)
```

```
        self.setLayout(main_layout)

    def set_boxes(self):
        global image_preview
        self.width_box.setText(str(image_preview.width))

    def on_height_change(self):
        logging.debug('on height change')

        try:
            h = int(self.height_box.text())
            self.apply_buttton.setEnabled(True)

            if(h < 0 or h > 1500):
                raise Exception()

            logging.debug('new height value: %d' % h)
        except:
            self.apply_buttton.setEnabled(False)
            logging.debug('invalid height value')

    def on_width_change(self):
        try:
            w = int(self.width_box.text())
            self.apply_buttton.setEnabled(True)

            if(w < 0 or w > 1500):
                raise Exception()

            logging.debug('new width value: %d' % w)
        except:
            self.apply_buttton.setEnabled(False)
            logging.debug('invalid width value')

    def on_apply(self):
        logging.debug('on apply')

        w = int(self.width_box.text())
        h = int(self.height_box.text())

        global image_preview, drawer, operations
        operation = operations.get_operation('resize')
        drawer.set_operation(image_preview, operation)
        drawer.process(image_preview, (w, h))
        self.parent.parent.refresh_image()
```

12. RotationTab – вкладка, яка відповідає за частину графічного інтерфейсу, де можна обертати зображення та відображати його.

ui.py

```
class RotationTab(QWidget):
```

```
    def __init__(self, parent):
        super().__init__()
        self.parent = parent

        rotate_left_buttton = QPushButton("↺ 90°")
        rotate_left_buttton.clicked.connect(self.on_rotate_left)

        rotate_right_buttton = QPushButton("↻ 90°")
        rotate_right_buttton.clicked.connect(self.on_rotate_right)

        flip_left_buttton = QPushButton("⇆")
        flip_left_buttton.clicked.connect(self.on_flip_left)

        flip_top_buttton = QPushButton("↑↓")
        flip_top_buttton.clicked.connect(self.on_flip_top)

        rotate_label = QLabel("Rotate")
        rotate_label.setAlignment(Qt.AlignCenter)
        rotate_label.setFixedWidth(140)

        flip_label = QLabel("Flip")
        flip_label.setAlignment(Qt.AlignCenter)
        flip_label.setFixedWidth(140)

        label_layout = QHBoxLayout()
        label_layout.setAlignment(Qt.AlignCenter)
        label_layout.addWidget(rotate_label)
        label_layout.addWidget(flip_label)

        buttton_layout = QHBoxLayout()
        buttton_layout.setAlignment(Qt.AlignCenter)
        buttton_layout.addWidget(rotate_left_buttton)
        buttton_layout.addWidget(rotate_right_buttton)
        buttton_layout.addWidget(flip_left_buttton)
        buttton_layout.addWidget(flip_top_buttton)

        main_layout = QVBoxLayout()
        main_layout.setAlignment(Qt.AlignCenter)
        main_layout.addLayout(label_layout)
        main_layout.addLayout(buttton_layout)

        self.setLayout(main_layout)

    def on_rotate_left(self):
        logging.debug('on rotate left')

        global image_preview, drawer, operations
        operation = operations.get_operation('rotate')
        drawer.set_operation(image_preview, operation)
        drawer.process(image_preview, 90)
        self.parent.parent.refresh_image()
        self.parent.modification_tab.set_boxes()
```

```
    def on_rotate_right(self):
        logging.debug('on rotate right')

        global image_preview, drawer, operations
        operation = operations.get_operation('rotate')
        drawer.set_operation(image_preview, operation)
        drawer.process(image_preview, 270)
        self.parent.parent.refresh_image()
        self.parent.modification_tab.set_boxes()

    def on_flip_left(self):
        logging.debug('on flip left')

        global image_preview, drawer, operations
        operation = operations.get_operation('flip')
        drawer.set_operation(image_preview, operation)
        drawer.process(image_preview, Image.FLIP_LEFT_RIGHT)
        self.parent.parent.refresh_image()

    def on_flip_top(self):
        logging.debug('on flip top')

        global image_preview, drawer, operations
        operation = operations.get_operation('flip')
        drawer.set_operation(image_preview, operation)
        drawer.process(image_preview, Image.FLIP_TOP_BOTTOM)
        image_preview.set_display_strategy(strategies['label'])
        self.parent.parent.refresh_image()
```

12. MainLaoyut – головна частина графічного інтерфейсу, яка акумулює у собі всі його складові. Чатина паттеру ”Стан” та “Заступник”, виступає у ролі об’єкту, доступ до якого треба обмежити.

ui.py

```
class MainLayout(QVBoxLayout):
    def __init__(self, parent):
        super().__init__()
        self.parent = parent
        self.file_name = None
        self.__licensed = None

        self.upload_buttton = QPushButton("Upload")
        self.upload_buttton.clicked.connect(self.on_upload)

        self.reset_buttton = QPushButton("Reset")
        self.reset_buttton.setEnabled(False)
        self.reset_buttton.clicked.connect(self.on_reset)

        self.save_buttton = QPushButton("Save")
```

```
    self.save_buttton.setEnabled(False)
    self.save_buttton.clicked.connect(self.on_save)

    buttton_layout = QHBoxLayout()
    buttton_layout.setAlignment(Qt.AlignTop)
    buttton_layout.addWidget(self.upload_buttton)
    buttton_layout.addWidget(self.reset_buttton)
    buttton_layout.addWidget(self.save_buttton)
    self.addLayout(buttton_layout)

    self.image_label = QLabel()
    self.image_label.setAlignment(Qt.AlignCenter)
    self.addWidget(self.image_label)
    global strategies
    strategies['label'] = LabelStrategy(self)

    self.action_tabs = ActionTabs(self)
    self.addWidget(self.action_tabs)
    self.action_tabs.setVisible(False)

    self.user_label = QPushButton()
    self.user_label.setStyleSheet(
       "QPushButton{font-size: 10px; color: grey;}")
    self.user_label.setFlat(True)
    self.user_label.clicked.connect(self.parent.user_authorize)
    self.addWidget(self.user_label)

def set_license(self, merchant, state):
    if (type(merchant) is not PediUI):
       raise Exception

    self.__licensed = state
       logging.debug('license set to %s', self.__licensed)

def on_upload(self):
    logging.debug('on upload buttton clicked')

    img_path, _ = QFileDialog.getOpenFileName(self.parent, "Open image",
       "/home/traumgedanken/Pictures",
       "Images (*.png *jpg)")
    self.file_name = ntpath.basename(img_path)

    global image_original, image_preview
    image_original = PediImage(img_path)
    image_preview = image_original.clone()
    self.refresh_image()

    self.reset_buttton.setEnabled(True)
    self.save_buttton.setEnabled(True)
    if (not self.__licensed):
       self.save_buttton.setToolTip('Sign in to unlock this')
    self.action_tabs.setVisible(True)
```

```
        self.action_tabs.adjustment_tab.reset_sliders()
        self.action_tabs.modification_tab.set_boxes()

    def on_reset(self):
        logging.debug('on reset buttton clicked')

        global image_original, image_preview
        image_preview = image_original.clone()
        self.refresh_image()

    def on_save(self):
        logging.debug('on save buttton clicked')

        new_img_path = LicensedState.run(
            self) if self.__licensed else NotLicensedState.run(self)

        if new_img_path:
            logging.debug("save output image to %s" % new_img_path)
            global image_preview
            image_preview.save(new_img_path)

    def refresh_image(self):
        global image_preview, strategies
        image_preview.set_display_strategy(strategies['label'])
        image_preview.show()
        image_preview.set_display_strategy(strategies['thumb'])
        image_preview.show()
        self.action_tabs.modification_tab.set_boxes()
```

12. PediUI – клас, який служить точкою входу в виконання програми. Член паттерну “Заступник”. В своєму конструкторі дає змогу користувачу авторизуватися. Член паттернів “Стан” та “Заступник”.

ui.py

```
class PediUI(QWidget):
    def __init__(self):
        super().__init__()

        self.main_layout = MainLayout(self)
        self.setLayout(self.main_layout)

        self.setMinimumSize(600, 500)
        self.setMaximumSize(900, 900)
        self.setGeometry(600, 600, 600, 600)
        self.setWindowTitle('pedi OOP2')
        self.center()
        self.show()

        self.user_authorize()
```

```
def user_authorize(self):
    user = None

    while(not user):
        login, ok_pressed = QInputDialog.getText(
        self, "SIGN IN", "Enter your login:", QLineEdit.Normal, "")
        if (ok_pressed):
            user = DataBase.get_user(login)
        else:
            break

    self.main_layout.user_label.setText(
        f'licenced to: {user.login}' if user else 'not licensed')
    self.main_layout.set_license(self, bool(user))

def center(self):
    qr = self.frameGeometry()
    cp = QDesktopWidget().availableGeometry().center()
    qr.moveCenter(cp)
    self.move(qr.topLeft())
```

2. ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ ЗА ДОПОМОГОЮ ШАБЛОНІВ ПРОЕКТУВАННЯ

2.1. Обґрунтування вибору та опис шаблонів проектування для реалізації програмного забезпечення автомату

1. Command

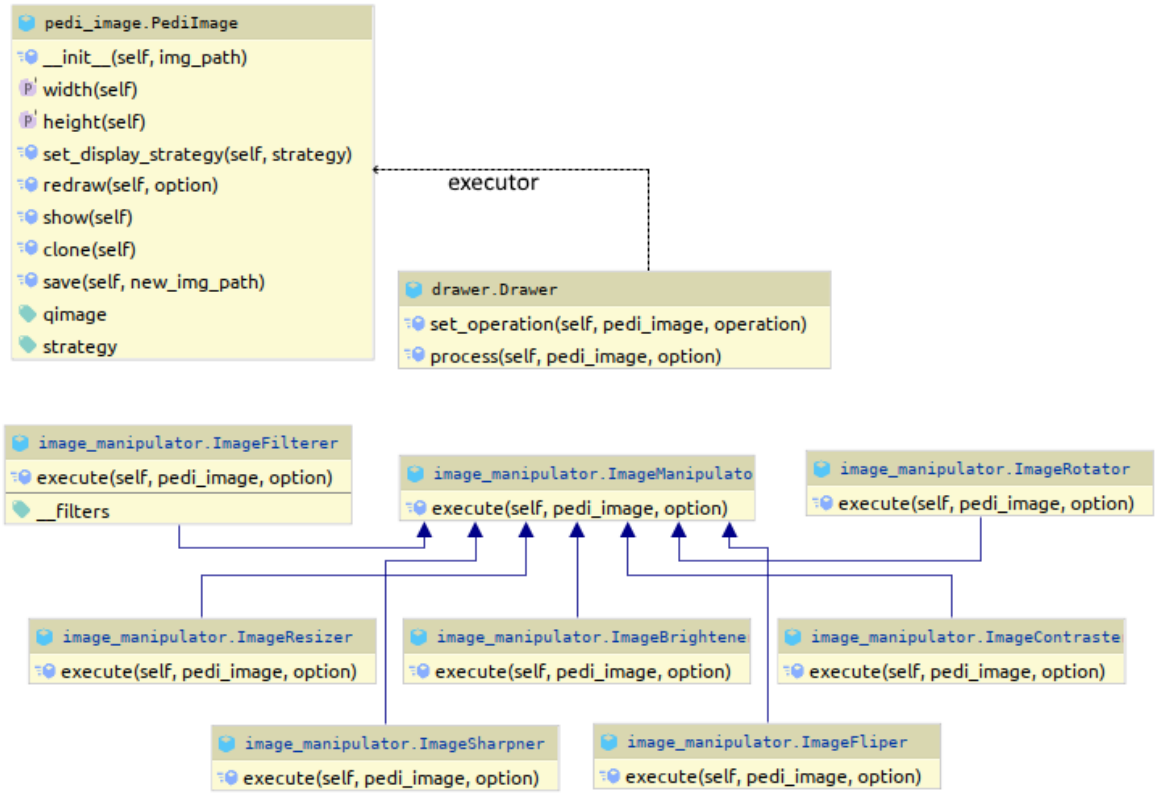

Рис. 2.1.1. UML-діаграма шаблону ***Command***

Поведінковий шаблон. Перетворює операцію в об'єкт. Це дозволяє передавати операції як аргументи при викликах методів, ставити операції в чергу, логувати їх, а також підтримувати можливість скасування операцій.

Структура. Реалізований для класу **PediImage**. Об’єкти цього класу мають у середині об’єкт **Operation**, і цей об’єкт слугує логічною частиною способу обробки зображення. Клас **Drawer** може застосувати операцію до забраження.

Обґрунтування використання даного шаблону. При написанні коду завжди слід прагнути розділення інтерфейсу та логіки. За допомогою формування такого класу і можна досягти інкапсуляції логіки в одному об’єкті.

2. Adapter

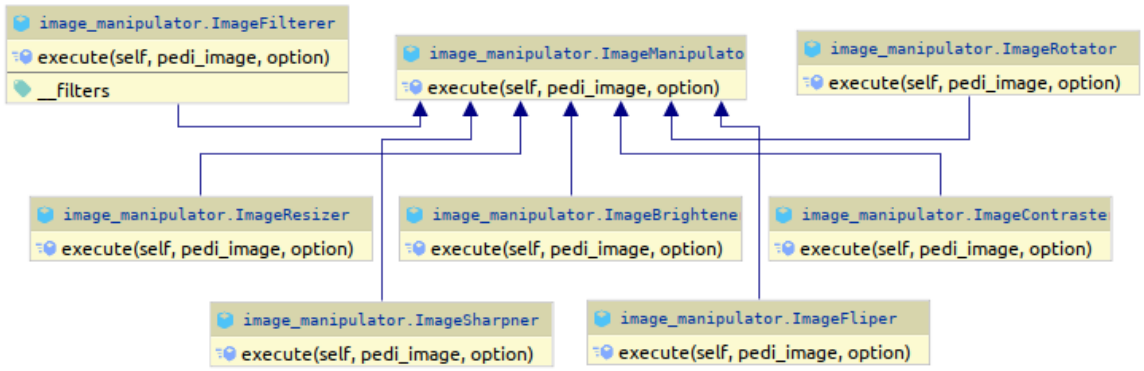

*Рис. 2.1.2. UML-діаграма шаблону "**Adapter**"*

Структурний шаблон. Забезпечує спільну роботу класів з несумісними інтерфейсами шляхом створення спільного об'єкта, через який вони можуть взаємодіяти.

Структура. Конкретні адаптери - **ImageResizer , ImageResizer, ImageRotator, ImageFliper, ImageFilterer, ImageBrightener, ImageContraster, ImageSharpner.**

Обґрунтування використання даного шаблону. Реалізований для того, щоб мати один інтерфейс, через який можна безпечно та зручно взаємодіяти зі всією системою операцій. Операції, які виконуються над зображеннями реалізовані різними способами, а деякі з них проводяться за допомогою посторонніх бібліотек, тому цілком доцільно було уніфікувати інтерфейс усіх операцій.

3. Прототип

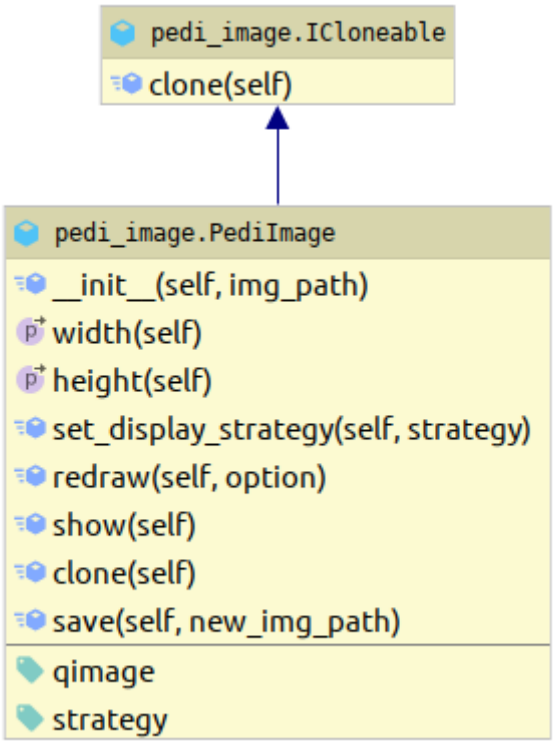

*Рис. 2.1.3. UML-діаграма шаблону "**Prototype**"*

Структурний шаблон. Задає види створюваних об'єктів за допомогою екземпляра-прототипу і створює нові об'єкти шляхом копіювання цього прототипу.

Структура. **ICloneable** – головний інтерфейс даного шаблону. **PediImage** — клас, конструктор якого приймає шлях до зображення, яке треба зчитати і має метод *clone()*, який дозволяє створити новий об'єкт зображення без звертання до файлової системи.

Обґрунтування використання даного шаблону. Створення зображення шляхом читання його з файлової системи набагато затратніше, аніж його створення шлях копіювання уже зчитаного зображення.

4. Proxy

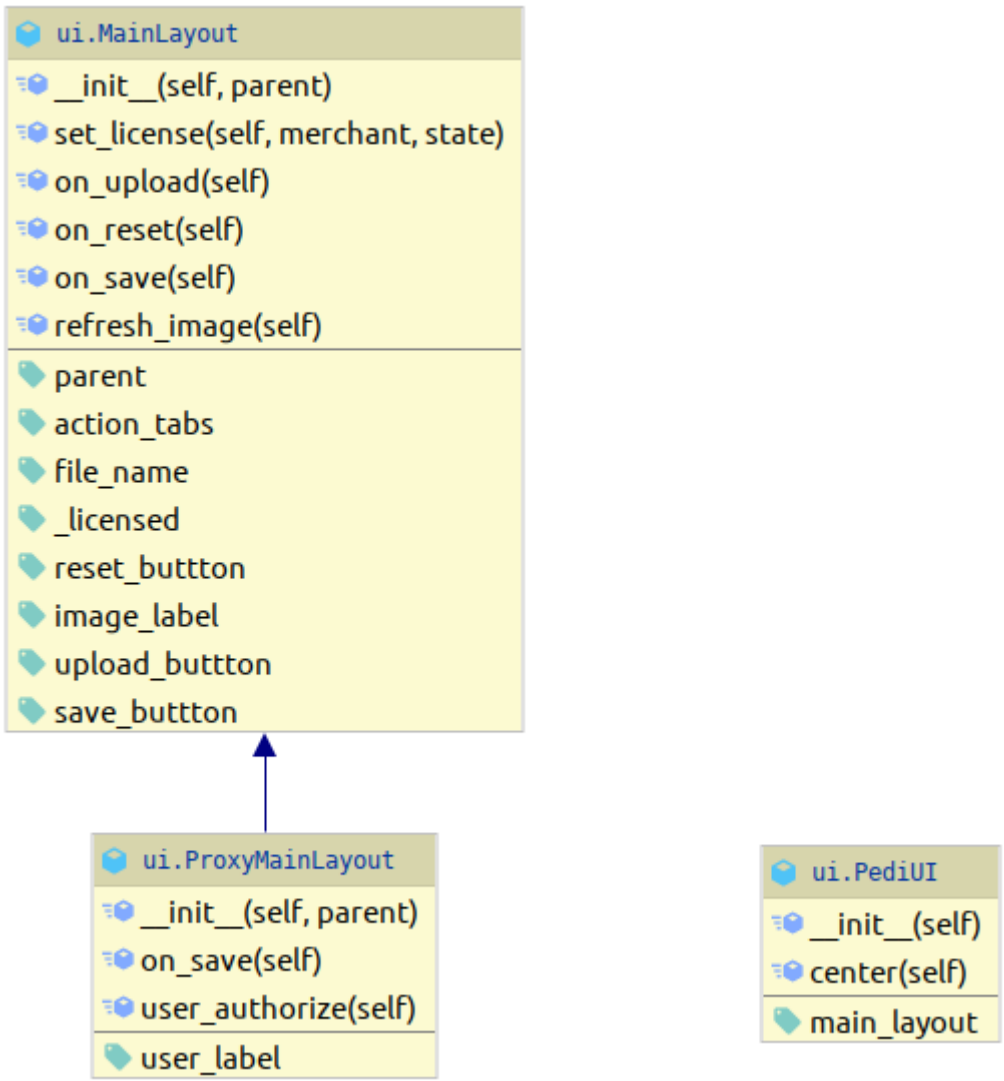

*Рис. 2.1.4. UML-діаграма шаблону "**Proxy**"*

Структурний шаблон. Обертає корисний об'єкт або сервіс спеціальним об'єктом-замінником, який "прикидається» оригіналом і перехоплює всі виклики до нього, а потім, після деякої обробки, направляє їх до обгорнутого об'єкту.

Структура. **MainLayout —** клас, стан якого змінює його поведінку. **PediUI —** його заступник, який огортає та містить поле типу **MainLayout** та імітує собою клас, для яйого є заступником.

Обґрунтування використання даного шаблону. **PediUI** робить вигляд, що є **MainLayout,** але на початку роботи змушує користувача авторизуватися.

5. State

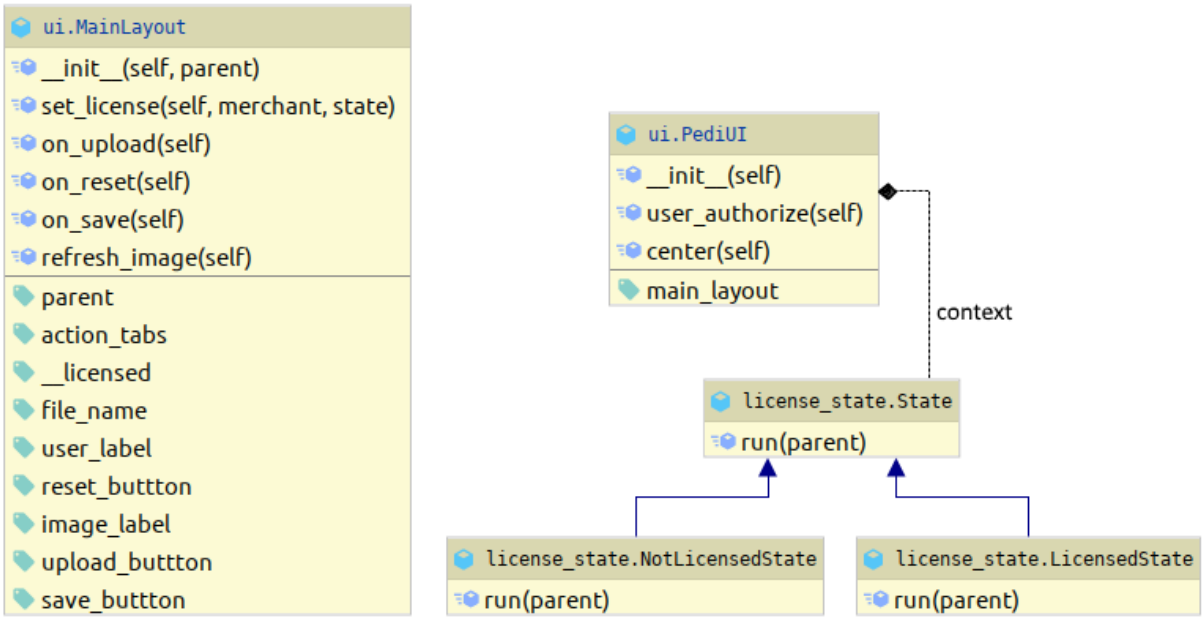

*Рис. 2.1.5. UML-діаграма шаблону "**State**"*

Поведінковий шаблон. Створює механізм підписки, за допомогою якого одні об'єкти можуть підписуватися на оновлення, що відбуваються в інших об'єктах.

Структура. **MainLayout** – основний компонент системи, поведінка якого залежить від його стану в поточний момент часу (ліцензія активована/не активована). **LicensedState** і **NotLicensedState** **LicensedState** класи, які описують поведінку в залежності від стану. **PediUI —** єдиний клас, який може змінювати стан.

Обґрунтування використання даного шаблону. У мові Python більша частина за приватність внутрішніх полів та методів класу лежить на відповідальності самих програмістів, тому було прийнято рішення розробити таку систему паттерну "Стан", де стан об'єкту **MainLayout** має змогу змінювати лише об'єкт **PediUI.**

6. Flyweight

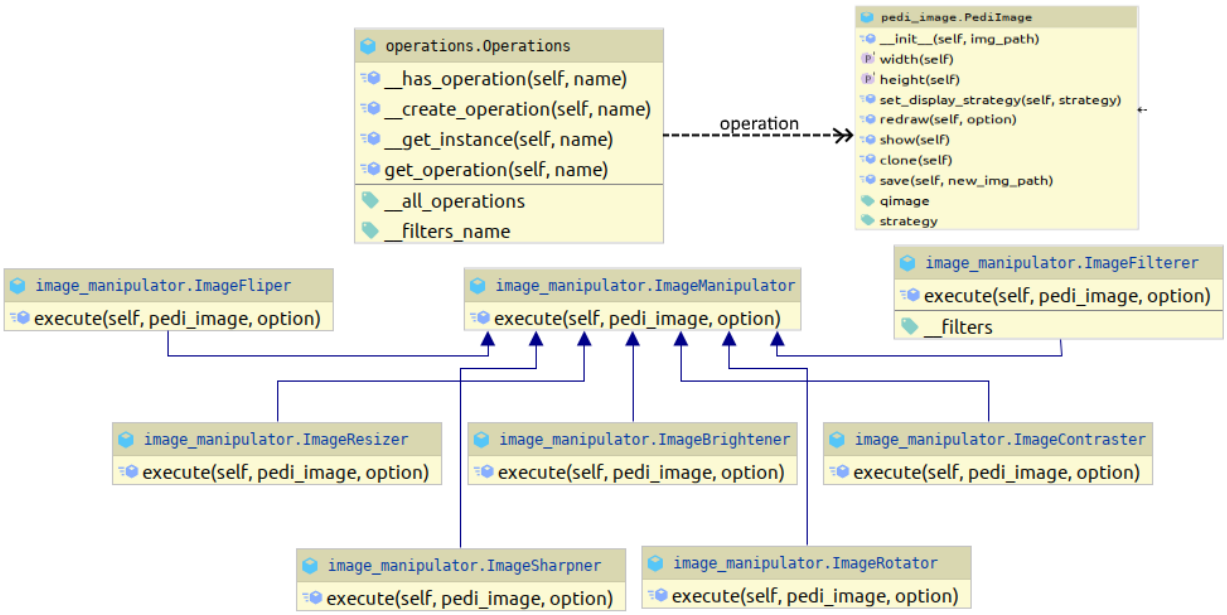

*Рис. 2.1.6. UML-діаграма шаблону "**Flyweight**"*

Структурний шаблон. Структурує об'єкти таким чином, що з них створюється лише обмежений набір екземплярів замість великої множини об'єктів. Полегшує повторне використання багатьох малих об'єктів, роблячи використання великої кількості об'єктів більш ефективною.

Структура. **NotifyMenu** – клас-обгортка для класу **Menu**, який водночас є представником підписника на події і декоратором відповідного класу. Об'єкт даного класу містить у собі об'єкт класу **Menu** (в тому числі – потенційно можливого меню, яке реалізовує інтерфейс **IPageMenu**) і переадресовує усі базові операції, які виконуються з меню, до представника, для якого він слугує "обгорткою". Додатково, цей клас розширює функціонал меню і дозволяє меню виступати у якості підписника на події.

Обґрунтування використання даного шаблону. В графічному редакторі доволі часто виконуються маніпуляції з зображеннями, тому недоцільно створювати окремий об'єкт операції кожного разу.

7. Strategy

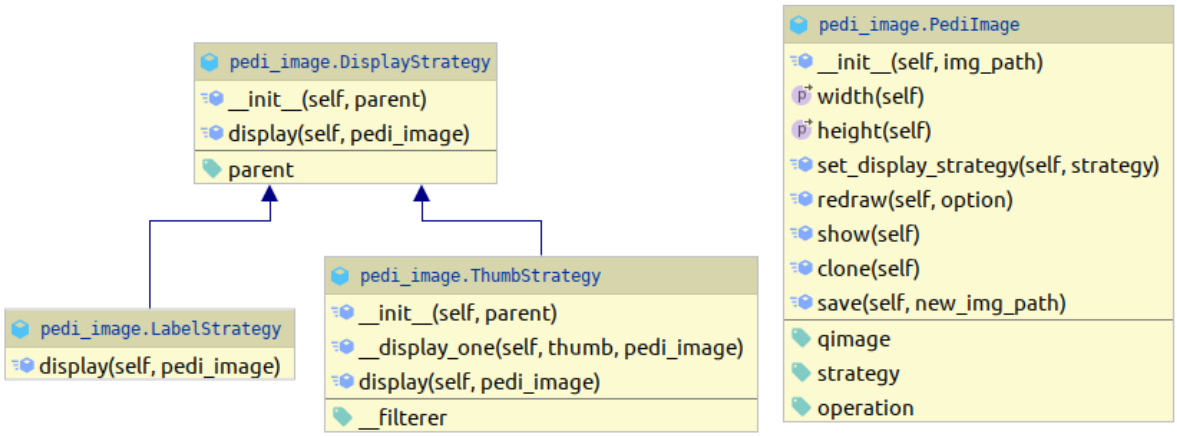

*Рис. 2.1.7. UML-діаграма шаблону "**Strategy**"*

Поведінковий шаблон. Визначає сімейство алгоритмів, інкапсулює кожен з них та робить їх взаємозамінними. Дозволяє змінювати алгоритми незалежно від коду клієнтів. Також відомий як Policy. Якщо в системі є алгоритми, які часто можуть використовуватися повторно в різних частинах програми, зручно їх виділити в окрему сутність, параметризувати та запускати там, де це потрібно, не дублюючи сам код.

Структура. **LabelStrategy** і **ThumbStrategy** — описують стратегії відображення зображення на екрані в різних місцях та в різних цілях. **PediImage** — має поле, яке відповідає за поле стратегії.

Обґрунтування використання даного шаблону. Алгоритми відображення зображення у різних місцях дуже сильно різняться між собою, тому доцільно було відокремити їх та інкапсулювати в методи з однаковими назвами.

2.2. Діаграма класів

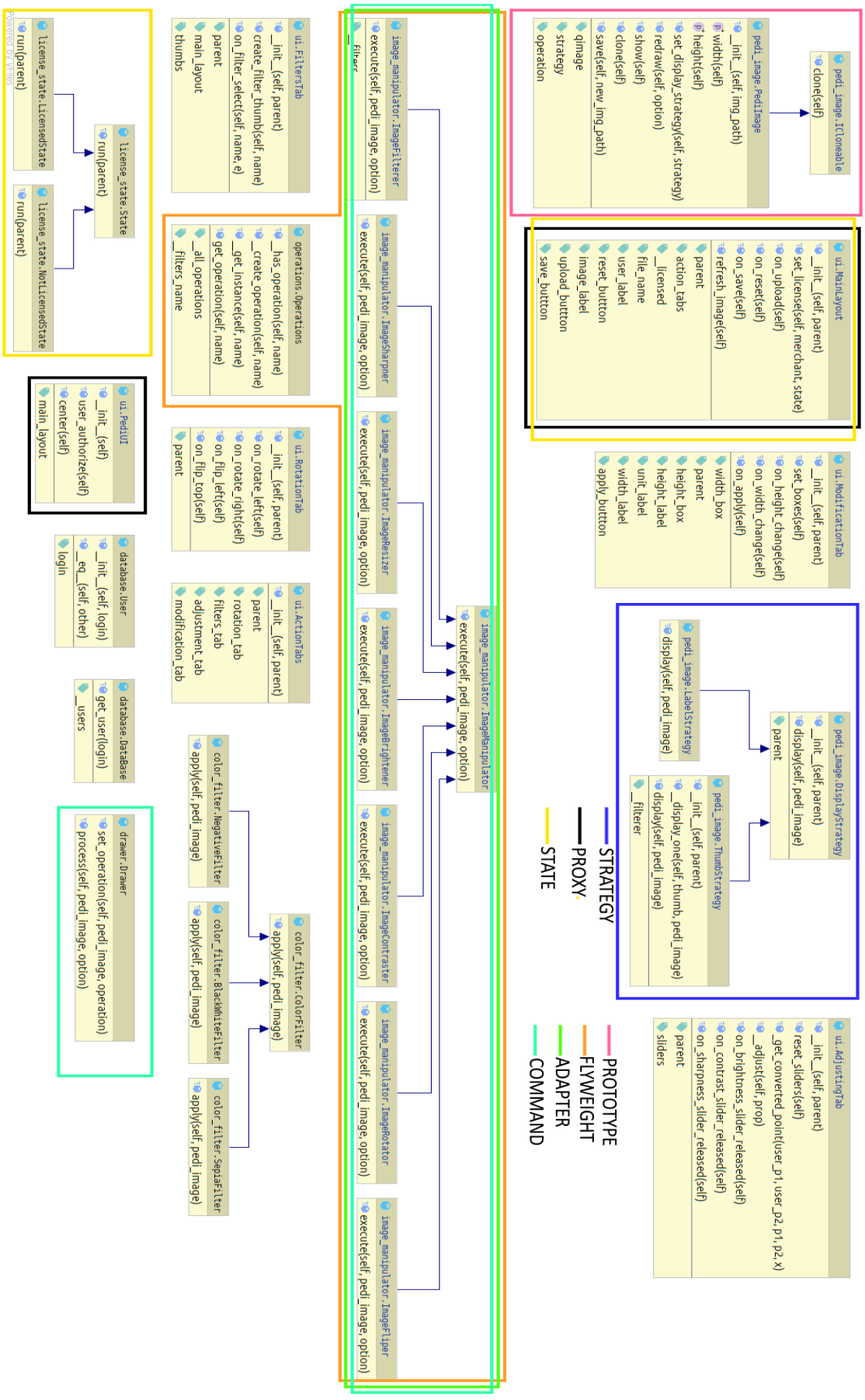

Рис. 2.2.1. Діаграма класів програми

2.3. Результати роботи програми

Початок роботи графічного редактору починається з вікна, де користувачу надається можливість авторизуватися:

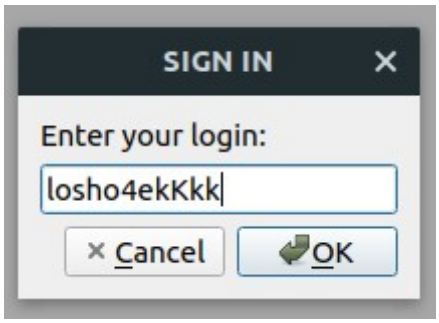

Рис. 2.3.1. Вікно авторизації

Після вибору зображення воно відображається на екрані:

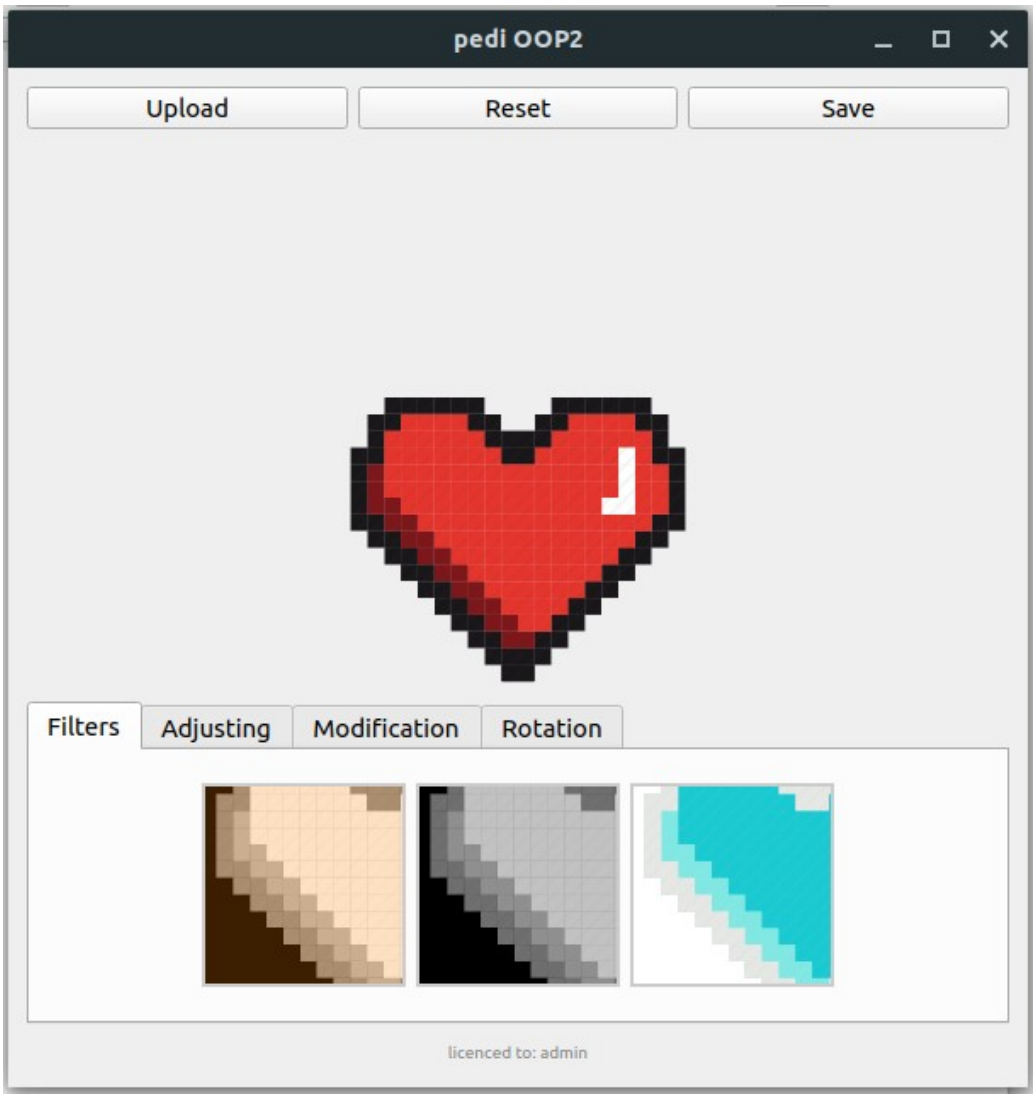

Рис 2.3.2. Відображення зображення на екрані (вкладка Filters)

На вкладці *Adjusting* є можливість змінити яскравість, контраст та чіткість:

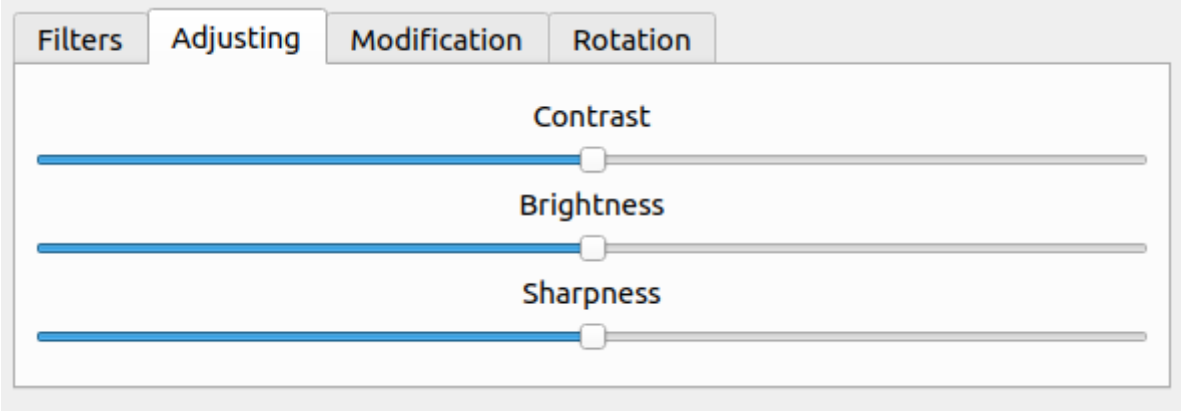

Рис 2.3.3. Вкладка Adjusting

На владці *Modification* можна змінити розмір зображеня у пікселях:

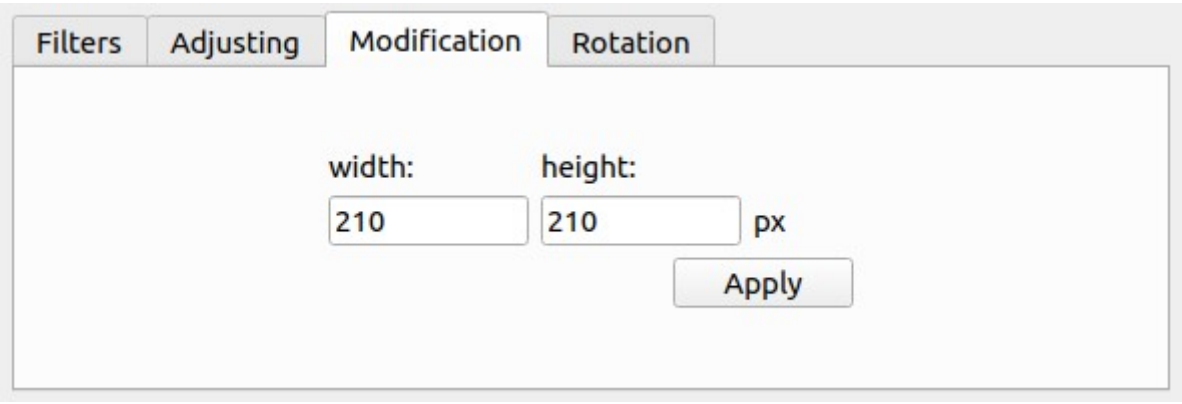

Рис 2.3.4. Владка Modification

На вкладці *Rotation* можна повернути/відобразити зображення.

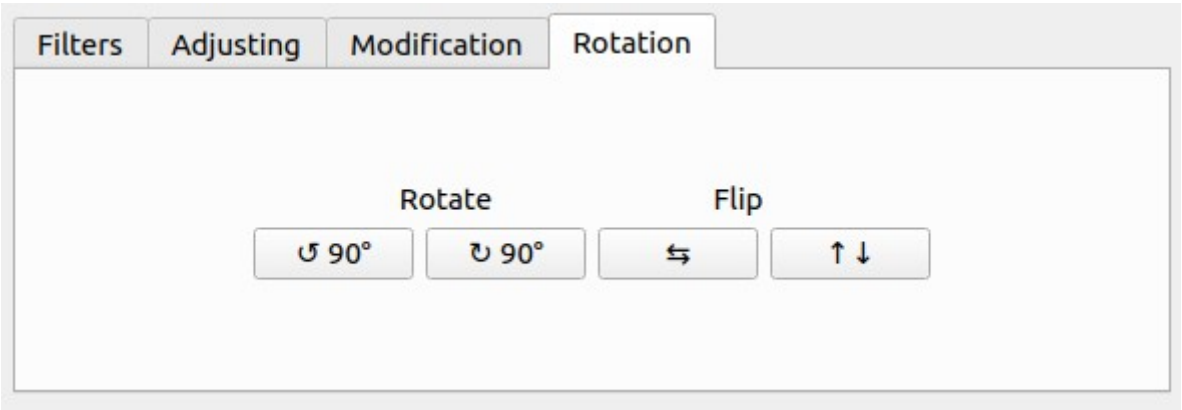

Рис 2.3.4. Вкладка Ratation

ВИСНОВКИ

Метою даної курсової роботи було розроблення програмного забезпечення графічного редактора з використанням шаблонів проектування. Підставою для розроблення стало завдання на виконання курсової роботи з дисципліни «Об’єктно-орієнтоване програмування» студентами ІІ курсу кафедри ПЗКС НТУУ «КПІ».

Для досягнення поставленої мети у повному обсязі виконано завдання, визначені у аркуші завдання на курсову роботу; розроблено графічні матеріали; реалізовано всі вимоги до програмного продукту, наведені у технічному завданні; створено відповідну документацію. Розроблене програмне забезпечення дозволяє користувачу редагувати зображення (різні кольорні фільтри, зміна розміру/яскравості/контрасту/чіткості, обертання/відображення).

Програму створено на основі використання шаблонів проектування. Зокрема, до структури програмного забезпечення входить реалізація семи шаблонів, які належать до різних груп.

Для розроблення програмного забезпечення була використана мова програмування Python.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Язык шаблонов. Города. Здания. Строительство. / Кристофер Вольфганг Александер. – 1977. – 1096 с.
2. Приёмы объектно-ориентированного проектирования. Паттерны проектирования / Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес. – 1994. – 395 с.
3. Руководство Microsoft по проектированию архитектуры приложений. / С. Сомасегар, Скотт Гатри, Девид Хилл. – 2009. – 529 с.